Maintaining, Minimizing, and Recovering Machine Language Program through SEU in On-Board Computer

Tomohiro Harada*, Keiki Takadama**

*The University of Electro-Communications, Japan
Research Fellow of the Japan Society for the Promotion of Science DC1
e-mail: harada@cas.hc.uec.ac.jp

**The University of Electro-Communications, Japan
e-mail: keiki@inf.uec.ac.jp

Abstract

This paper explores the feasibility of Tierra-based OBC that evolves programs through a bit inversion caused by Single-Event Upset. In particular, Tierra-based OBC is applied to the evolution of the PIC assembly programs. Additionally, this paper proposes a new method that can reliably recover the broken program to the correct one. Intensive experiments reveal that (1) Tierra-based OBC cannot only maintain the PIC machine language programs, but also can minimize the program size and the execution step of the PIC machine language program through SEU, (2) Tierra-based OBC *with* the proposed recover method can maintain and evolve the PIC assembly program in high SEU rate in comparison with one *without* the recover crossover, and (3) even if all of the correct PIC programs are broken from a memory, Tierra-based OBC *with* the proposed recover method can recover the correct programs by modifying the broken programs.

1 Introduction

The bit inversion called *Single-Event Upset (SEU)* [14] of semiconductor devices such as a memory or CPU occurs when the semiconductor devices are exposed to space radiation. SEU causes software error of a spacecraft. SEUs and *multi-bit upsets (MBU)* were, for example, observed in DRAM and SRAM on commercial semiconductor devices (CSD) on the mission demonstration test satellite-1 (MDS-1, TSUBASA) developed by JAXA [4]. To overcome this problem, the conventional approaches employed (1) shielded devices, (2) multiplex logic circuit, or (3) CPUs with a thick process rule. However, a huge cost is required with approaches (1) and (2) because they increase satellite weight or require much space. While an approach (3) makes CPU calculation ability low. Typical software checksum mechanisms [6] also used, however, they are hard to correct for more than one bit inversion, i.e., MBU [14].

As a novel approach toward SEU, our previous study proposed Tierra-based On-Board Computer (OBC) [11, 3, 2] that enables to *evolve* the computer programs through the bit inversion caused by SEU. This approach is based on the idea of Tierra [7, 13, 12] where digital creatures (implemented by a program) are evolved through a mutation in a gene, and by employing the technique of Genetic Programming [8]. An advantage of Tierra-based OBC is to generate small size program by the evolution. This means that Tierra-based OBC can generate the program that has robustness to SEU by using the bit inversion of SEU since small size program has low probability to be affected by SEU. However, the current version of Tierra-based OBC has the following problems: (1) Tierra-based OBC can only evolve simple programs which includes simple four arithmetic operations, but it is not investigated to evolve more complex programs like machine language programs including branch structures and many registers; and (2) Tierra-based OBC can maintain the correct programs and minimize the program size, but it cannot recover the correct programs when the correct programs are broken caused by SEU if a complexity of program increases.

To tackle these problems, this paper aims at (1) exploring the feasibility of Tierra-based OBC to evolve programs written in an actual machine language by applying Tierra-based OBC to the evolution of actual machine language programs for space mission, and (2) proposing a novel method to reliably recover the correct program from the broken program in addition to maintain the correct programs and minimize both the program size and the step. Note that the minimization of the program size contributes to decrease the probability to be affected by SEU, while the minimization of the step contributes to improve the speed of processing. Toward this aim, this paper employs an instruction set that can be used on PIC [10] developed by Microchip Technology Inc.. This is because the PIC micro-controller is carried on many kind of spacecraft and it can execute programs for space mission. The

PIC instruction set consists of 33 basic instructions (addition, subtraction, logic operations, bit operations, and branch instructions) composed of 12bits and has 16 general purpose registers and one working register which are composed of 8bits. The essential differences between four arithmetic operations program and the PIC program are (1) four arithmetic operations program is executed sequentially, while the PIC program needs to consider the structure of the programs (e.g., loop or branch), and (2) four arithmetic operations program only employs two registers, while the PIC program needs to handle 17 registers (16 general purpose registers and one working register), which cause the difficulty of the program.

To investigate the feasibility of Tierra-based OBC to evolve the PIC program and the effectiveness of the proposed method to recover the correct program, this paper conducts the following experiments in Tierra-based OBC: (1) evolving the PIC programs that execute numerical calculation and the Boolean calculation; and (2) recovering programs from the situation where the correct programs are broken during the evolution process. In both experiments, Tierra-based OBC with/without the proposed recovering method are compared.

This paper is organized as follows. Section 2 explains Tierra-based OBC, and Section 3 points its problems. Section 4 describes the proposed recovering method. Section 5 conducts experiments and shows their results and finally Section 6 concludes this paper.

2 Tierra-based On-Board Computer

2.1 Overview

Tierra-based On-Board Computer (OBC) is an OBC that embeds the mechanism of *Tierra-based Asynchronous Genetic Programming (TAGP)* [11, 3, 2] we proposed. TAGP is based on the idea of Tierra [13] that is a biological evolution simulator, where the digital creatures are evolved through a cycle of a self-reproduction, a deletion and genetic operators such as a crossover, a mutation or an instruction insertion/deletion. TAGP improves Tierra to be able to evolve programs that can solve any given tasks from the engineering point of view, unlike Tierra can only evolve the programs, i.e., digital creatures, that aim at reproduce themselves. TAGP introduces *fitness* commonly used in Evolutionary Algorithms such as Genetic Algorithms (GAs) [1] or Genetic Programmings (GPs) [8] to evaluate the programs, i.e., the programs are reproduced or deleted according to their *fitness* values. Mean square error or error ratio is usually employed as the fitness function.

Fig. 1 shows an illustration of TAGP. TAGP starts from programs that completely solve the given engineering task (aim). These programs consist of some instructions with some registers and they are stored in a limited

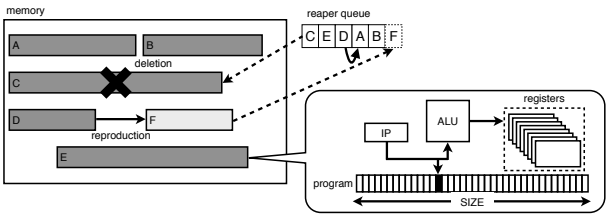

Figure 1. An illustration of TAGP

memory space (population). All programs are inserted in the *reaper queue* that controls their lifespan. They execute a few instructions in turn for a parallel execution. When all instructions in one program are completely executed, its fitness is calculated depending on its calculation result, and the program is asynchronously reproduced according to its fitness value. When the memory is filled with the programs, the programs located at the upper position in the reaper queue are removed from the memory.

2.2 Algorithm

The algorithm of the current version of TAGP [11, 3, 2] is described in Algorithm 1. TAGP can evolve programs for a given task through the following (1) selection and reaper queue control, (2) reproduction, and (3) deletion procedures. In this algorithm, $prog$, $prog.f$, and $prog.f_{acc}$ respectively indicate the program currently executed, its fitness, and its accumulated fitness. *pre-prog* indicates the program previously executed. f_{max} indicates the maximum value of the fitness, while $rand(0, 1)$ indicates the random real value between 0 and 1. P_{down} and P_{up} are described later.

(1) Selection and reaper queue control

When one program is completely executed, its fitness is calculated, and the calculated fitness is added to an accumulated fitness $prog.f_{acc}$ (line 1 in Algorithm 1). If the accumulated fitness of the program exceeds f_{max}, it is selected as a parent, and f_{max} is subtracted from its accumulated fitness (lines 2 and 3). Note that a program having a high fitness has a high probability to be selected as a parent because the accumulated fitness frequently exceeds f_{max}, while a program having a low fitness is hard to satisfy this condition.

After that, the position in the reaper queue of the program selected as the parent become lower than the current one, *i.e.*, its deletion probability decreases (which means to survive long) (lines 4-6). While the position of the program not selected as a parent becomes upper, *i.e.*, its deletion probability increases (which means to be easily removed) (lines 20-22). The distance to move lower/upper is determined by the probability P_{down} and P_{up} which are calculated as the following equation based on fitness of

Algorithm 1 The algorithm of TAGP

```
1:  prog.f_acc ← prog.f_acc + prog.f
2:  if prog.f_acc ≥ f_max then
3:      prog.f_acc ← prog.f_acc − f_max
4:      repeat
5:          down position of prog in reaper queue
6:      until rand(0, 1) < P_down(prog.f)
7:      generate an offspring by mating prog with pre-
        prog through genetic operators
8:      pre-prog ← prog
9:      delete the program located the most upper in reaper
        queue
10:     if prog.f = f_max then
11:         if prog is better than elite_prev then
12:             reproduce (copy) prog without any genetic
                operators
13:         else
14:             generate an offspring by mating prog with
                pre-prog through genetic operators
15:         end if
16:         elite_prev ← prog
17:         delete the program located the most upper in
            reaper queue
18:     end if
19: else
20:     repeat
21:         up position of prog in reaper queue
22:     until rand(0, 1) < P_up(prog.f)
23: end if
```

the program,

$$P_{down}(f) = \frac{f}{f_{max}} \times P_r,\ P_{up}(f) = \frac{f_{max} - f}{f_{max}} \times P_r, \qquad (1)$$

where P_r is the maximum probability of P_{down} and P_{down}, which is predetermined. For example, the fitness is calculated as f_{max}, P_{down} is calculated as P_r which is the maximum probability. In this case a lot of down process is executed and its position in the reaper queue changes close to the lowest in the reaper queue.

(2) Reproduction

To reproduce the program asynchronously, TAGP generates an offspring by mating *prog* with *pre-prog* through the genetic operators such as a crossover, a mutation, and an instruction insertion/deletion operators (line 7). The crossover operator combines *prog* with *pre-prog*. The mutation operator changes one random instruction in *prog* to other random instruction. The insertion operator inserts one random instruction into *prog*, while the deletion operator removes one instruction selected at random from it.

Additionally, to preserve programs which can completely execute the given task, the elite preserving strategy [5] is applied (lines 10-18). Concretely, if $prog.f$ is calculated as f_{max}, which means it completely execute the given task, it is selected as an *elite*. It is compared with the previous *elite* represented as $elite_{prev}$. Then if *prog* is better than $elite_{prev}$, it is reproduced (copied) as an elite program *without* genetic operators. While even if not, it generates an offspring by mating with *pre-prog with* genetic operators.

Figure 2. Tierra-based OBC (Flight Model)

Table 1. Components in Tierra-based OBC

MCU	H8/3069 (25MHz)×2
RAM	DRAM (16MBit) ×2
ROM	EEPROM (256KBit) ×6
weight	197g

(3) Deletion

TAGP conducts a deletion operator when an offspring is generated (lines 9 and 17). Concretely, the program located at upper in the reaper queue is removed. Such program is commonly aged and having a low fitness.

2.3 Hardware architecture

We constructed Tierra-based OBC in our previous research [2] with two H8s as the micro controller unit (MCU) and DRAMs as shown in Figure 2 and Table 1, and it is carried on SHIN-EN (UNITEC-1) [15]. H8 is the microprocessor developed by Renesas Technology that is constructed on one chip with a CPU core, internal memory, timers, and I/O ports. Tierra-based OBC consists of two H8/3069s (25MHz), two DRAMs (16MBit), and 6 EEPROMs (256KBit), and it weighs 197g. Programs on the Tierra-based OBC are stored in DRAM, and H8 executes TAGP processes — processes that should not change. From this reason, H8/3069 is employed because it is robust against space radiation and many university satellites with H8/3069 successfully operate on actual space missions. Ordinary DRAM is used because programs are expected to be evolved by bit inversion. We confirmed that the process of TAGP is successfully executed on this OBC architecture.

3 Problems

Our previous researches [11, 3, 2] reported that the current version Tierra-based OBC can evolve simple programs that include simple four arithmetic operations and uses a few registers even if random bit inversions occur in program memory space and register values during the evolution. However, the current version of Tierra-based OBC has the following problems: (1) it is not investigated to evolve more complex programs like machine language programs including branch structures and many registers; and (2) Tierra-based OBC can maintain the correct programs and minimize the program size, but it cannot recover the correct programs when the correct programs are broken caused by SEU if a complexity of program increases.

For the first problem as evolution of complex programs, to apply Tierra-based OBC to actual space missions, it is necessary to evolve *machine code level* programs, in other words, assembly language programs. Toward this aim, this paper explores the feasibility of Tierra-based OBC to evolve programs written in an actual machine language by applying Tierra-based OBC to the evolution of actual machine language programs for space mission. In particular, we employ the PIC [10] assembly language, which is developed by Microchip Technology Inc.. The PIC instruction set consists of 12 bits 33 instructions including addition, subtraction, Boolean logic, bitwise operations, and branch instructions. A program can use any of 16 general purpose registers and one register (named *working register*). Each register consists of 32bits. Since PIC micro controller is usually used in actual space missions, it is possible to apply Tierra-based OBC to actual space missions if the PIC assembly programs can be evolved.

For the second problem as recovering the correct programs, it is desired for Tierra-based OBC to maintain or evolve programs even if the correct programs are broken caused by SEU. It is, however, hard to recover the correct program in such situation in the current version of Tierra-based OBC. For this reason, this research proposes a novel method to reliably recover the correct program from the broken programs. The detailed algorithm is described in Section 4.

4 Proposed method: Recover crossover

4.1 Overview

To recover broken programs through an evolution process of Tierra-based OBC, this paper proposes a novel method named as *recover crossover (RX)*. In SEU situation, it is assumed that the programs in Tierra-based OBC cannot calculate the correct result due to only a few *wrong* instructions in them, but not almost of instructions, and it can be recovered if these wrong instructions are replaced with the correct ones. While it is also assumed that, in comparison with the similar size programs, almost part of program have same instructions, but an instruction at a point of a wrong instruction may differ from each other. From this assumption, it is indicated that an appropriate assembling of programs has the possibility to generate the correct program. In particular as shown in Figure 3, in comparison of the similar size programs, the broken programs have a few wrong instructions, and an instruction at a point of wrong instruction differs from each other. Therefore, the correct program can be generated by exchanging the instructions at the point of different instructions.

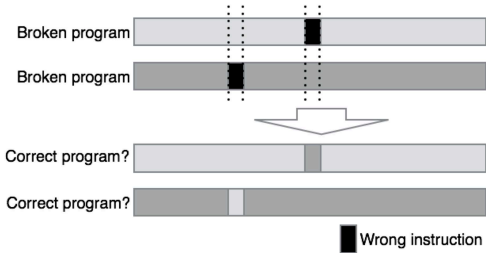

Figure 3. Recovering the correct program from the broken programs

4.2 Levenshtein distance

To achieve this recovering method, it is necessary to measure the difference between two programs. This paper employs the *Levenshtein distance* [9] to measure the difference between two programs. The Levenshtein distance (otherwise known as *edit distance*) is originally developed to measure the difference two strings or sequences by considering single character insertion/deletion/substitution as single operator and counting the minimum number of operations from one string/sequence to another one. For example, considering two sequences "ABCDE" and "ACDFG", three operators are required to convert the sequence "ABCDE" to "ACDFG", in particular, the deletion of the 2^{nd} "B", the substitution of the 5^{th} "D" to "F", and the insertion of "G" to the end of the sequence as Figure 4. From this calculation, the Levenshtein distance between these two sequences is calculated three, and representing three operators, insertion, deletion, and substitution as [i], [d], and [s] respectively, the convert from "ABCDE" to "ACDFG" is represented as "A[d]CD[s][i]".

4.3 Algorithm

The recover crossover considers a program as a sequence of instructions, and calculates the operators to convert a broken program to another one that is selected from the population. And then, the recover crossover selects

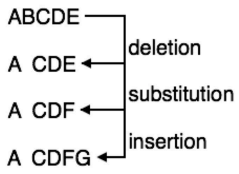

Figure 4. An example how to calculate the Levenshtein distance

Algorithm 2 A flow of the recover crossover (inserted between step 19 and step 20 in Algorithm 1)

1: select a program which size is similar to *prog*
2: calculate the Levenshtein distance between two programs
3: select a few operators and apply them to *prog*

a few operators (insertion, deletion, or substitution) calculated by the Levenshtein distance, and applies them to a broken program. This operation converts a broken program close to another selected one. For example described above, considering two programs, "ABCDE" and "ACDFG", if the 2^{nd} deletion ([d]) operator is selected, the sequence "ABCDE" is converted to "ACDE" by applying the deletion operator, which is closer to the another sequence "ACDFG" than the previous program.

The detailed flow of the recover crossover is described in Algorithm 2, and its image is illustrated in Figure 5. The recover crossover is executed for the broken program in the process of TAGP. The broken program is determined whether its accumulated fitness exceeds the maximum fitness or not. If not, it cannot accomplish the given task that should be recovered. To recover the broken program, the program that has similar program size to the broken program is selected from the population, and the Levenshtein distance is calculated regarding these two programs. Then a few operators are selected and they are applied to the broken program. For example in Figure 5, three operators are calculated with the Levenshtein distance and if [s] is

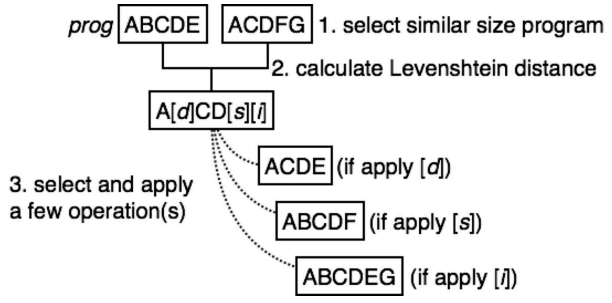

Figure 5. An illustration of the recover crossover

Table 2. Testbed programs

Numerical			
ID	function	Init. size	Init. step
N1	$f(x) = x^4 + x^3 + x^2 + x$	62	715
N2	$f(x, y) = x^y$	25	708
N3	$f(x) = x^5 - 2x^3 + x$	61	714
N4	$f(x) = x^6 - 2x^4 + x^2$	61	714
Even-parity			
E1	5bits-Parity	15	15
E2	6bits-Parity	17	17
E3	7bits-Parity	19	19
E4	8bits-Parity	21	21

selected, a new program that has a sequence "ABCDF" is generated. This flow is inserted between step 19 and step 20 in Algorithm 1.

5 Experiments

5.1 Cases

To investigate the feasibility of Tierra-based OBC to evolve the PIC assembly program and to investigate the effectiveness of the proposed method, the recover crossover, this paper conducts the following two experiments in Tierra-based OBC:

- **Case1:** evolves the PIC programs that execute the numerical calculation and the even-parity calculation.
- **Case2:** recovers the PIC programs from the situation where the correct programs are broken during the evolution process, i.e., all programs in Tierra-based OBC cannot calculate the correct results

In both cases, programs that executes numerical calculations or the even-parity calculation shown in Table 2, and Tierra-based OBC with/without the recover crossover are compared. Afterword, Tierra-based OBC *without* the recover crossover is represented as TOBC, while one *with* the recover crossover is represented as TOBC/rx.

5.2 Settings

Table 3 summarizes parameters used in this experiment. The unit time is defined to be able to execute 100 instructions, while the different SEU rates are tested to investigate the robustness of Tierra-based OBC to high SEU rate. In Case2 where all correct programs are broken through the evolution, one instruction of all programs change to another random instruction and all programs cannot calculate the correct result at half of the maximum execution time (1.0×10^8 unit time).

Table 3. Parameter settings

Population size	20
Crossover rate	0.7
Mutation rate	0.1
Insertion rate	0.1
Deletion rate	0.1
#executable insts.	100insts./unit time
SEU rate	$10^{\{-3,-4,-5,-6\}}$/unit time
Execution time	2.0×10^8

The following fitness function are respectively employed for the numerical and the even-parity problems:

$$f_N = \frac{f_{max}}{\frac{1}{n}\sum_{i=1}^{n}\left|\hat{y}_i - y_i^*\right|} \quad (2)$$

$$f_E = \frac{f_{max}}{\frac{2}{n}\sum_{i=1}^{n}\delta(\hat{y}_i, y_i^*)}, \delta(x,y) = \begin{cases} 1 & x = y \\ 0 & x \neq y \end{cases}, \quad (3)$$

where $\hat{y}_i$ indicates the i^{th} output value of a program, while y_i^* indicates the i^{th} target value.

The experiment conducts 30 independent runs for each cases and each SEU rate, and we evaluate (1) a success rate that is calculated from the number of the runs that successfully maintain the correct programs, (2) average size of program generated after finishing evolution, and (3) average execution steps of program generated after finishing evolution. Note that criteria (2) and (3) are given by the best program in each run.

5.3 Result: Case1

The experimental results are shown in Table 4, and 5. From these results, it is indicated that Tierra-based OBC can evolve all kind of the PIC assembly program, even though it is more complex than the programs that is evolved in the previous researches. In particular, Tierra-based OBC does not only maintain the correct programs through the bit inversion of SEU, but also minimizes the program size and the number of execution steps except for the problem N2. While Tierra-based OBC *without* the recover crossover cannot maintain the correct programs in some cases where the SEU rate is high as $1-^{-3}$/unit time. Concretely, the programs for the numerical problems cannot be maintained in some cases. This is because the programs for the numerical problems include loop structure that can be easily broken due to the bit inversion. For this reason, it is hard to maintain such programs with the current version of Tierra-based OBC. Note that the SEU rate as 10^{-3}/unit time is very high rate, so that such situation is hardly observed in the actual space mission.

On the other hands, Tierra-based OBC *with* the recover crossover can maintain the correct programs even in high SEU rate, but two runs in N2. This result reveals that the recover crossover contributes to increase the ability to maintain the correct programs of Tierra-based OBC. The result of the average program size and the average number of the execution step supports the recover crossover that cannot only maintain the correct programs but also minimize the program size and the execution step, like the current version of Tierra-based OBC.

Table 6. Success rate of TOBC in Case2

	10^{-3}	10^{-4}	10^{-5}	10^{-6}
N1	0.03	0.10	0.00	0.03
N2	0.03	0.00	0.10	0.03
N3	0.03	0.07	0.00	0.03
N4	0.03	0.03	0.03	0.03
E1	0.23	0.23	0.03	0.07
E2	0.17	0.10	0.03	0.13
E3	0.10	0.07	0.13	0.00
E4	0.07	0.07	0.10	0.03

Table 7. Success rate of TOBC/rx in Case2

	10^{-3}	10^{-4}	10^{-5}	10^{-6}
N1	0.93	0.87	0.93	0.87
N2	0.90	1.00	0.90	0.93
N3	0.87	0.87	0.90	0.93
N4	0.93	0.93	0.97	0.93
E1	1.00	1.00	1.00	0.83
E2	0.93	0.97	1.00	0.90
E3	0.93	0.93	0.97	0.87
E4	1.00	0.90	0.93	0.90

5.4 Result: Case2

At first, the success rate in Case2 are shown in Table 6 and 7. This result indicates that Tierra-based OBC *without* the recover crossover hardly maintain the correct programs in all testbeds, even if the SEU rate is low. This reveals that the current version of Tierra-based OBC cannot recover the correct programs if they disappear from the population. On the other hand, Tierra-based OBC *with* the recover crossover successfully maintains (recovers) the correct programs without a few runs. From these results, it is revealed that the recover crossover works well to recover the correct programs from the broken programs.

Table 8 and 9 shows the average program size and the average execution steps in Tierra-based OBC *with* the recover crossover. Note that this result only averages the program size and the execution step in runs where the correct programs are successfully recovered. This result indicates that Tierra-based OBC *with* the recover crossover continues to evolve programs after recovering the correct programs, i.e., continues to minimize the program size and the execution step.

Table 4. Results of TOBC in Case1

	(1)Success rate				(2)Ave. size				(3)Ave. execution steps			
	10^{-3}	10^{-4}	10^{-5}	10^{-6}	10^{-3}	10^{-4}	10^{-5}	10^{-6}	10^{-3}	10^{-4}	10^{-5}	10^{-6}
N1	0.87	1.0	1.0	1.0	45.5	48.8	46.4	46.1	838.7	746.9	800.8	828.8
N2	0.93	1.0	1.0	1.0	35.4	41.0	35.0	31.6	750.8	777.2	703.8	677.9
N3	1.0	1.0	1.0	1.0	48.3	49.2	50.4	47.8	794.8	784.8	852.4	822.0
N4	0.90	1.0	1.0	1.0	50.1	48.1	48.0	47.2	752.7	826.2	866.5	818.2
E1	1.0	1.0	1.0	0.97	11.7	11.7	11.6	11.4	14.5	14.3	14.9	12.8
E2	1.0	1.0	1.0	1.0	14.5	14.8	14.9	14.9	19.1	18.7	19.6	18.5
E3	1.0	1.0	1.0	1.0	17.6	17.9	17.8	17.3	23.1	23.8	25.1	23.8
E4	1.0	1.0	1.0	1.0	20.0	20.2	20.1	20.1	28.2	25.6	26.9	26.4

Table 5. Results of TOBC/rx in Case1

	(1)Success rate				(2)Ave. size				(3)Ave. execution steps			
	10^{-3}	10^{-4}	10^{-5}	10^{-6}	10^{-3}	10^{-4}	10^{-5}	10^{-6}	10^{-3}	10^{-4}	10^{-5}	10^{-6}
N1	1.0	1.0	1.0	1.0	50.4	50.2	47.1	47.0	838.7	746.9	800.8	828.8
N2	0.93	1.0	1.0	1.0	31.3	42.8	25.6	30.6	750.8	777.2	703.8	677.9
N3	1.0	1.0	1.0	1.0	49.4	48.1	49.6	48.2	794.8	784.8	852.4	822.0
N4	1.0	1.0	1.0	1.0	45.6	47.4	45.8	45.9	752.7	826.2	866.5	818.2
E1	1.0	1.0	1.0	1.0	11.6	11.7	11.9	11.3	14.5	14.3	14.9	12.8
E2	1.0	1.0	1.0	1.0	14.7	14.7	14.9	15.0	19.1	18.7	19.6	18.5
E3	1.0	1.0	1.0	1.0	17.5	17.4	17.6	17.6	23.1	23.8	25.1	23.8
E4	1.0	1.0	1.0	1.0	19.8	20.0	20.0	20.0	28.2	25.6	26.9	26.4

6 Conclusion

This paper has explored the feasibility of Tierra-based OBC to evolve the PIC assembly programs and the effectiveness of the proposed method to *recover* the correct program from the broken one. In particular, the proposed recover crossover modifies the broken program by combining it with other program. We have conducted the experiments that evolve PIC assembly programs with several SEU rate, and the following implications have been revealed: (1) Tierra-based OBC cannot only maintain the PIC machine language programs, but also can minimize the program size and the execution step of the PIC machine language program through SEU; (2) Tierra-based OBC *with* the recover crossover can maintain and evolve the PIC assembly program in high SEU rate in comparison with one *without* the recover crossover; and (3) even if the correct PIC programs are broken in a population, Tierra-based OBC *with* the recover crossover can recover the correct programs by modifying the broken programs.

What should be noticed here is that the experimental results have revealed that the recover crossover cannot completely recover the broken programs. Therefore, further improvement of the recover crossover and other recover method should be explored in near future. Such important directions must be pursued in the near future in addition to the following future research: (1) a verification on the evolution of a program that has more complex instruction set or more registers such as H8 micro controller; (2) an exploration of the robustness to the bit inversion in a core process of Tierra-based OBC for example an error of an evaluation process; and (3) a construction of Tierra-based OBC on hardware component and a verification on it.

7 Acknowledgements

This work was supported by Grant-in-Aid for JSPS Fellows Grant Number 249376.

References

[1] D. E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1 edition, 1989.

[2] T. Harada, M. Otani, H. Matsushima, K. Hattori, H. Sato, and K. Takadama. Robustness to Bit Inversion in Registers and Acceleration of Program Evolution in On-Board Computer. *Journal of Advanced Computational Intelligence and Interlligent Informatics (JACIII)*, 15(8):1175–1185, 10 2011.

[3] T. Harada, M. Otani, H. Matsushima, K. Hattori, and K. Takadama. Evolving Complex Programs in Tierra-based On-Board Computer on UNITEC-1. In *International Astronautical Congress (IAC), 2010 61st World Congress on*, 2010.

Table 8. Average program size in TOBC/rx in Case2

	10^{-3}	10^{-4}	10^{-5}	10^{-6}
N1	47.8	47.7	47.3	45.0
N2	34.9	33.6	32.3	25.7
N3	53.5	49.3	49.3	50.0
N4	50.9	54.4	48.0	46.3
E1	11.7	11.8	11.7	11.7
E2	14.6	14.7	14.6	14.8
E3	17.6	17.7	17.4	18.0
E4	19.9	19.9	19.9	19.9

Table 9. Average execution steps in TOBC/rx in Case2

	10^{-3}	10^{-4}	10^{-5}	10^{-6}
N1	796.8	812.2	869.3	804.0
N2	682.1	771.6	673.1	665.4
N3	784.4	821.5	788.3	818.2
N4	794.2	789.9	818.5	738.0
E1	14.7	14.8	13.4	14.7
E2	18.9	18.0	18.4	17.7
E3	23.4	22.4	21.9	25.1
E4	27.6	25.8	27.1	27.7

[4] N. IKEDA, H. SHINDOU, Y. IIDE, H. ASAI, S. KUBOYAMA, and S. MATSUDA. Evaluation of the Errors of Commercial Semiconductor Devices in a Space Radiation Environment. *The transactions of the Institute of Electronics, Information and Communication Engineers. B*, 88(1):108–116, 2005.

[5] D. Jong and K. Alan. *An Analysis of the Behavior of a Class of Genetic Adaptive Systems*. PhD thesis, Department of Computer and Communications Sciences, University of Michigan, 1975.

[6] J. Justesen and T. Hoholdt. *A Course In Error-Correcting Codes*. European Mathematical Society, 2004.

[7] T. Kimezawa and T. S. Ray. Artificial Life System Tierra. Technical Report TR-H-268, ATR Human Information Processing Research Laboratories, 1999.

[8] J. Koza. *Genetic Programming On the Programming of Computers by Means of Natural Selection*. MIT Press, 1992.

[9] V. I. Levenshtein. Binary codes capable of correcting deletions, insertions and reversals. *Soviet Physics Doklady*, 10(8):707–710, feb 1966. Doklady Akademii Nauk SSSR, V163 No4 845-848 1965.

[10] Microchip Technology Inc. *PIC10F200/202/204/206 Data Sheet 6-Pin, 8-bit Flash Microcontrollers*. Microchip Technology Inc., 2007.

[11] K. Nonami and K. Takadama. Tierra-based Space System for Robustness of Bit Inversion and Program Evolution. In *SICE, 2007 Annual Conference*, pages 1155–1160, 2007.

[12] T. S. Ray. An approach to the synthesis of life. *Artificial Life II*, XI:371–408, 1991.

[13] T. S. Ray. Documentation for the Tierra Simulator. http://life.ou.edu/pubs/doc/index.html, 1991.

[14] Reed Business Information. EDN Japan February issue. *EDN Japan*, 2005.

[15] UNISEC. UNITEC-1. http://www.unisec.jp/unitec-1/, 2009.